Invariant Performance: A Statement of Task Isolation Useful for Embedded Application Integration

Matthew M. Wilding, David S. Hardin, and David A. Greve

Rockwell Collins, Inc.
Advanced Technology Center
Cedar Rapids, IA 52498 USA
{mmwildin, dshardin, dagreve}@collins.rockwell.com

Abstract. We describe the challenge of embedded application integration and argue that the conventional formal verification approach of proving abstract behavior is not useful in this domain. We introduce *invariant performance*, a formulation of task isolation useful for application integration. We demonstrate invariant performance by formalizing it in the logic of PVS for a simple yet realistic embedded system.

1 Introduction

Integration of multiple real-time embedded applications onto a single processor is increasingly attractive because the capacity of computing devices continues to grow. The use of fewer devices reduces space and power consumption that can be very valuable in an embedded environment, and fewer device connections increase reliability. Greater integration can also simplify the development of fault-tolerant architectures.

Integration of applications poses daunting challenges as well, because integrated applications may interact. Applications that share computing resources can interfere with each other's *space* — values saved in memory by an application — and *timing* — the rate at which an application performs.

The system developer who wants to integrate an application with other applications has several concerns.

safety Can any other application in the system effect an application's performance to cause it not to meet its requirements?

security Can other applications in the system glean information that should be restricted to one application?

verification level Must each application be verified at the confidence level associated with the most critical application in the system?

verification completeness Has the verification of each application taken into account the many ways other applications might interfere with it?

These challenges can be met if the host computer system provides an encapsulation mechanism that separates applications so that they can be verified separately. This kind of mechanism, known as a "partitioning" system in the avionics community, allows not only the verification of integrated applications initially but can eliminate the need for reverification in future system configurations. An encapsulation mechanism must be no less trusted than the most trusted application in the system, so it is natural to turn to formal verification to gain a high level of confidence.

Operating system correctness statements in the literature have proved inadequate for application integration, so we have developed our own that we call *invariant performance*. In this paper we describe some previous work by others and introduce our notion of task isolation that allows separate verification of applications. We describe a small operating system and underlying hardware and a correctness statement that supports encapsulation useful for separate verification of integrated applications.

2 Verified Operating Systems

2.1 Verified Computer Systems

Machine-checked computer system proof has been used to build extremely reliable computer systems. Some examples of these involve compiled routines from the C string library targeted to the Motorola 68020 [5], microcode for the Motorola CAP processor [6], a stack of verified systems [3], verification of the "oral messages" algorithm [4, 10], code for some simple real-time systems [18], floating-point microcode [6,15], a verified Piton [13] program [17], floating-point hardware [16], a simple scheduler [7], and partial microcode correctness of some Rockwell Collins microprocessors [11,12].

Broadly speaking, each of these projects relates the execution of a model of a computer system with the execution of a more abstract model that describes the expected behavior of the system. This approach has also been used to establish aspects of operating system correctness. The most applicable example of the formal verification of an operating system is Bevier's verification of KIT [1,2].

2.2 KIT

KIT ("kernel, isolated, tasks") is an operating system kernel that enforces partitioning among a set of user tasks. The kernel supports some simple communication services for interpartition messages and input devices. KIT is implemented with about 300 instructions of machine code for a machine called TM, a fictional machine with a simple von Neumann architecture. TM has some hardware support useful for building a partitioning operating system kernel, including

- a memory protection scheme involving base and limit registers,
- user and executive modes that effect instruction execution,
- input and output ports that provide a communication mechanism, and
- a clock that decrements after each instruction and generates an interrupt when it reaches 0.

KIT provides a set of user-mode tasks with the illusion that each is operating in isolation, except for the effect of interprocess communication. Each I/O device is associated with exactly one partition. Most of the verification work to establish partition isolation involves showing that KIT correctly maintains the state of the various tasks in the face of partition swapping and task execution, and that it does the right thing when there is communication with I/O devices or between partitions.

The correctness theorem relates a very detailed and realistic model of KIT executing on TM to an abstract model where tasks execute in isolation. The KIT correctness statement embodies in microcosm the appeal of formal methods. Rather than attempt to list all the things that could go wrong in a partitioned operating system (a hopeless task for all but the simplest of systems) and show that nothing undesirable occurs, we demonstrate correct functioning of the system. Once this specification is proved we need not worry that we checked that a partition might overwrite another partition's memory space by, say, adding a value using an indirect address.

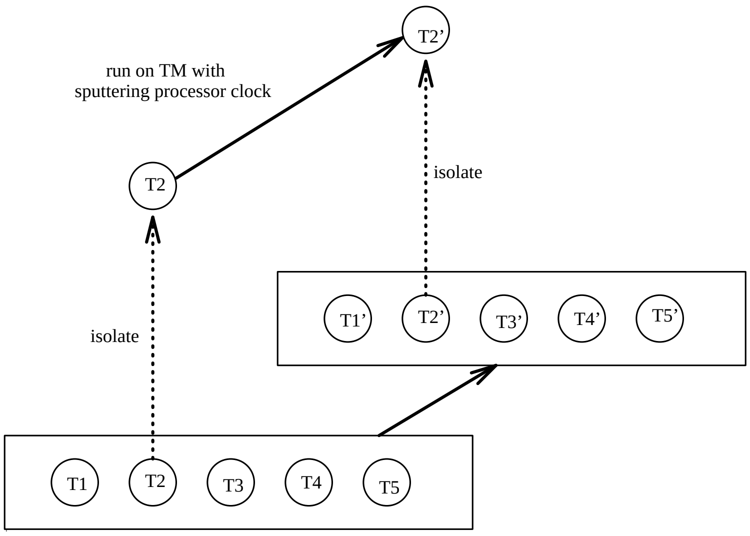

Fig. 1. The KIT correctness theorem[2]

The KIT correctness theorem relates the execution of each partition running in isolation on an unloaded TM to its execution under KIT, and is illustrated in Figure 1. The partition running on the abstract machine used to specify KIT's behavior executes in spurts that correspond to the allocation of the CPU by the kernel at the more concrete level, and this relationship is formalized in the statement of the correctness theorem.

2.3 Invariant performance

The KIT theorem — and most of the other verification projects with which we are familiar — provides a service guarantee about a computer system. Real-time concerns are not considered in the KIT verification, but the approach used by KIT could be extended to provide guarantees about throughput and latency.

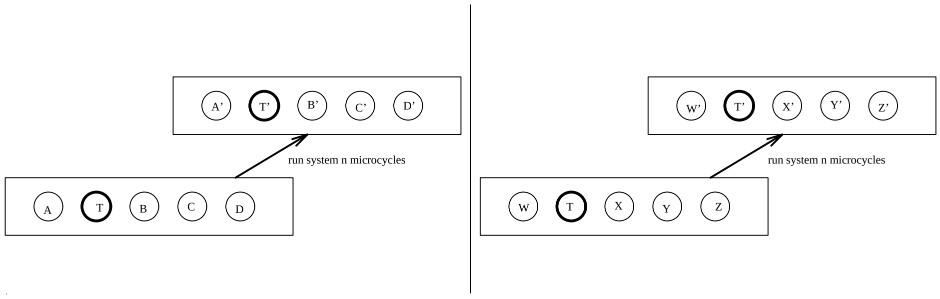

Fig. 2. Invariant Performance

But these properties do not suffice to allow separate verification of integrated applications. *It is unrealistic to expect application developers to verify applications against an abstract, unrealized machine model, even if the abstraction arguably characterizes "good" system behavior.* Our specification dispenses with the definition of the abstract machine altogether. We will use machine-checked proof to relate the execution of an application on a realistic model of the computer system with the application's execution when other applications are different. This approach is suggested by Figure 2: partition **T** operates without regard to the operation of the other partitions. If two kernel-controlled partition sets have identical initial kernel states and identical states for some partition **T** then the execution of the partition sets maintains the equivalence of the states of **T**.

The guarantee invariant performance makes to the application developer is that after his application is integrated with other applications it will work precisely as it worked before integration. If for example the developer tests his application in a partition he can rely on the system to work identically even if other partitions are used to host other applications. This can not be done, for example, with the KIT correctness statement: the developer does not know precisely how his application will run, only that it will run just like it would run on an abstract machine described by the correctness theorem. The abstract machine used in the KIT specification is like the concrete machine except that the clock sputters in a way that corresponds to CPU allocation under KIT.

We believe that invariant performance is the crucial property needed to allow separate verification of real-time, embedded applications since it provides complete time and space isolation. It also provides the strict separation needed to host applications that require isolation from each other in order to satisfy a security policy. It provides the application developer a development platform that will be unaffected by the application's integration, thereby allowing for independent verification of real-time, embedded applications.

3 An Example Application of Invariant Performance

We demonstrate invariant performance by developing its statement on a simple, concrete system. The system we model contains an AAMP-FV microprocessor [11] and a partition management unit ("PMU"). The PMU maintains memory isolation among the partitions and includes timers that allow temporal control of partitions by the kernel. The model of the AAMP-FV we use is adapted from the instruction-level AAMP-FV "macrolevel" model [11].

In this section we describe *Schultz*, a simple partitioning system, our formalization of invariant performance for Schultz, and an outline of the proof.

3.1 Schultz Overview

Schultz supports a cyclic schedule of noninterfering partitions. There is no explicit provision in this initial model for kernel-mediated shared resources such as interrupt controllers. Each partition is allocated a predetermined amount of CPU time with no interruption and has its own memory space, thereby ensuring noninterference between partitions. The Schultz kernel is very simple since there is only one shared resource, the CPU, and is responsible for initializing the system and maintaining the partition schedule. An unusual aspect of Schultz is microcycle-accurate partition-starting to achieve strict temporal independence among partitions.

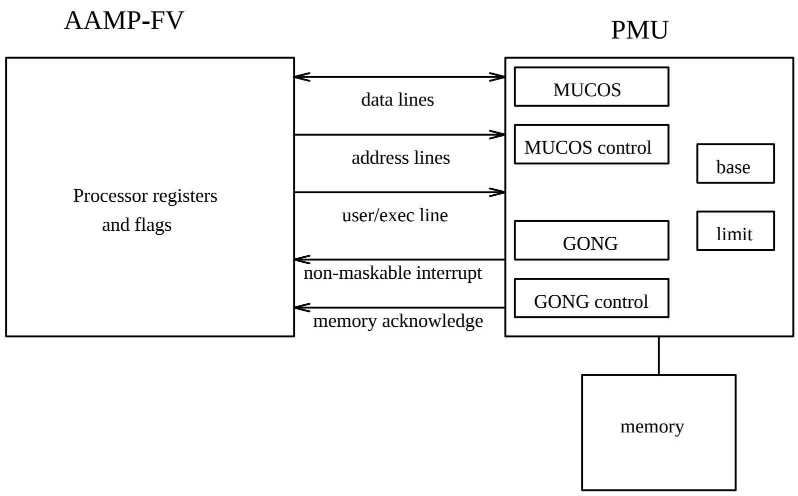

Fig. 3. Schultz Hardware

Figure 3 represents the Schultz hardware configuration. The AAMP-FV and the PMU interact in several ways.

- The accessible memory range is determined by the PMU using the processor user/executive line and base/limit addresses contained in memory-mapped PMU registers.
- For each processor memory transaction the PMU determines whether the location is currently accessible.
- A PMU timer, the *gong timer*, generates a periodic non-maskable interrrupt ("NMI") to the processor.
- A PMU timer, the *mucos timer*, is used to generate a memory acknowledge signal to the processor when the processor writes a non-zero value to a particular memory address, the *mucos register*. As will be discussed below, this mechanism provides for strict temporal synchronization.
- Each PMU timer has a corresponding memory-mapped control register that determines whether the counter is active and, if so, the value to which the timer is set when it expires.

Our AAMP-FV model is a modification of the model of [11]. We have modified the model in order to simplify underlying proofs about code following the suggestions of [19], particularly the use of an interpreter style to facilitate code proof automation.

The Schultz PMU maintains information about the current partition executing on the processor, including how long its CPU allocation is, how much CPU time it has used, and the range of memory to which it has access. Expiration of the "gong" timer signals the end of the current partition's CPU allocation. The "mucos" timer signals the end of partition switch handling time and is used to time the start of a partition, thereby synchronizing the timing of partition execution to eliminate any effect of unpredictable interrupt latency — interrupts are only recognized on instruction boundaries — or early partition exits due to illegal instruction execution. The timers are free-running timers that are decremented each microcycle. When a counter reaches 0 it is reinitialized with a corresponding timer initialization value. A timer is shut off by setting the timer initialization value to 0.

The two PMU timers operate differently, reflecting how they are used. The gong timer is used to signal partition switching and generates an NMI when it reaches 0. The mucos timer is used to synchronize the start of a partition to the correct microcycle and delays the memory acknowledge signal on non-zero writes to the mucos register until the mucos timer reaches 0.

The PMU also maintains a base/limit pair for the currently operating partition that is used to restrict partition access to memory. When a user-mode instruction writes to memory outside the range of addresses between the base and limit pair it has no effect, and when an address from outside the range is read it returns a value of 0.

We express the model and its correctness conjecture in the logic of the PVS theorem prover [14] since we intend to prove that Schultz provides invariant performance, using PVS to machine-check the proof. The PVS formalization of the

```
step(system): system_state =
  LET conns = conns(system), s = processor(system), p = pmu(system) IN

  % run next instruction with protected memory and PMU-generated NMI
  LET s2 = step_processor(s WITH [(mem):=protectmem(memory(conns),p,conns),
                                  (intreg):=setint6(intreg(s), nmi(conns))]) IN

  % calculate time of instruction execution
  LET itime = step_time(s,mucosset(s2),val(mucostimermod(p))>0,val(mucostimer(p))) IN

  % update connections to reflect PMU/processor operation
  LET conns2 = (# (nmi) := intreg(s2)^6 OR
                          (val(gongtimermod(p))>0 AND val(gongtimer(p)) <= itime),
                (memory):=resetmucos(restoremem(mem(s2),mem(s),p,conns)),
                (um) := um(s2),
                (itime) := itime #) IN

  % update PMU state
  LET p2 = step_pmu(p,memory(conns2),itime) IN

    make_system (s2, p2, conns2)

protected_system(system,cur,finish): RECURSIVE system_state =
  IF finish<=cur
   THEN system
   ELSE
     LET system2 = step(system) IN
       protected_system (system2, cur + itime(conns(system2)), finish)
   ENDIF
  MEASURE max(0,finish-cur)
```

Fig. 4. Schultz Hardware Formalization

Schultz hardware is presented in Figure 4. We have not provided enough information for the reader to understand this model fully. (The subsidiary functions are defined in our model but not presented in this paper, and we do not attempt here to describe the semantics of the PVS logic.) We present Figure 4 to convey that we have modelled the Schultz hardware in detail to support reasoning about it.

4 The Schultz Kernel

We now introduce a partition switch signal handler that implements partitioning. The AAMP-FV assembly code is given in Figure 5. The loop initializes the PMU and schedules the next partition in the schedule.

The scheduling code maintains several data structures. A schedule of partitions is assumed at symbolic location `psds`. The schedule is saved using a list of length `length` of process state descriptors ("PSD"s), each of which represents a partition. An AAMP-FV PSD is 8 double words containing processor state values PC, TOS, LENV, and PAGE and 4 unused locations. Schultz uses 2 of the spare PSD double words to maintain the base and limit values for each partition.

The symbolic value `curr` is the number of the current partition in the schedule. Each time a partition switch occurs we increment `curr`, thereby advancing

```
ELOOP:  LIT16 0FFFFh                ;
        ASN24 exec_code             ;

        REF24 length                ; if at end of schedule, reset pointer
        REF24 curr                  ;
        SUB                         ;
        SKIPT8 rpart                ;
        LIT4 0                      ;
        ASN24 curr                  ;

rpart:  LIT24 psds                  ; store psd table location
        REF24 curr                  ;
        DUP                         ; while we've got it, set PMU num reg
        ASN24 partn                 ;
        LIT4 4                      ; multiply curr pointer by 16 to get offset
        SHL                         ;
        LIT4 0                      ; make single word a double word
        ADDD                        ; calculate PSD pointer for next partition

        DUP                         ; set PMU base and limit from PSD
        LIT32 8                     ;
        ADDD                        ;
        DUP                         ;
        REFDA                       ;
        ASND24 breg                 ;
        LIT32 2                     ;
        ADDD                        ;
        REFDA                       ;
        ASND24 lreg                 ;

        LIT4 1
        LIT24 mreg
        ASNA                        ; write to mucos register

        LIT4 7                      ; clear pending interrupts - for example,
        CLRINT                      ; an NMI that occurred since illegal instr

        USER                        ; start user partition

        ASN24 exec_code             ; save returned value - could be partition
                                    ; signal or illegal instruction

        REF24 curr                  ; increment schedule pointer
        LIT4 1                      ;
        ADD                         ;
        ASN24 curr                  ;

        LIT16 ELOOP                 ; calculate skip value to loop


        SKIP                        ;
```

Fig. 5. Partition Switch Kernel Code

to the next partition in the schedule. The schedule is a simple, cyclic schedule, so when the value of `curr` reaches `length` it is reset to 0.

The partition switch handler uses the AAMP-FV USER instruction to start the next partition. Before executing the USER instruction, however, the handler reads the mucos register and clears pending interrupts.

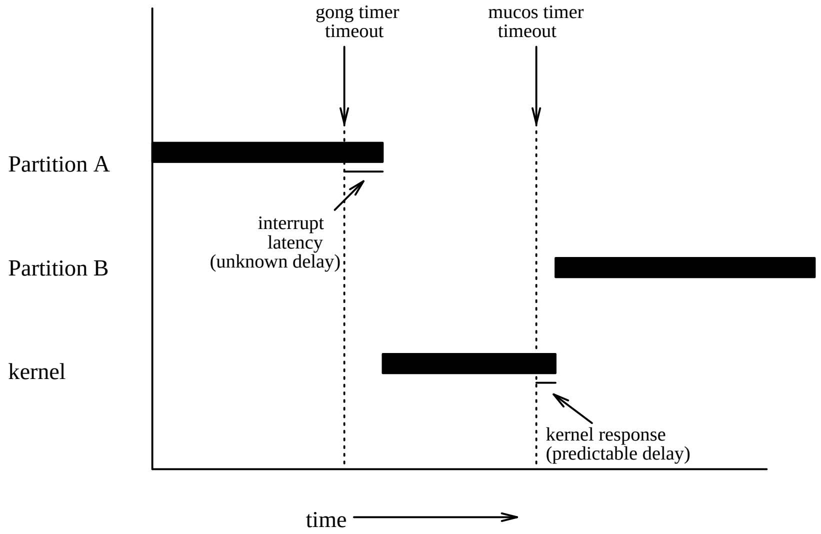

Fig. 6. A Typical Schultz Partition Switch

Strict time partitioning requires that partition scheduling be unaffected by what individual partitions do. This requires some effort in the face of interrupt latency, since interrupts like the partition swap interrupt are only recognized on instruction boundaries. The partition switch handler uses the PMU mucos register to guarantee strict time partitioning. Before starting a partition the mucos register is referenced. The memory transaction acknowledgement signal for the mucos register address is delayed until the mucos timer expires. Thus, the start of the next partition is delayed until a predictable time, as illustrated in Figure 6.

The handler also addresses another scenario: it is possible that the previous partition ended not because of a partition switch signal but rather due to the partition's execution of an illegal instruction, as illustrated in Figure 7. If this occurs then during execution of the handler the partition swap signal will be generated; and since the handler runs in executive mode the partition switch interrupt will be pending. If this happens it will occur before the mucos timer times out (since the mucos timer is designed to time out after the gong timer) so the partition switch interrupt will be pending after the mucos synchronization. The kernel executes a CLRINT instruction to clear this pending interrupt, thereby ensuring that the next partition receives its entire CPU allocation.

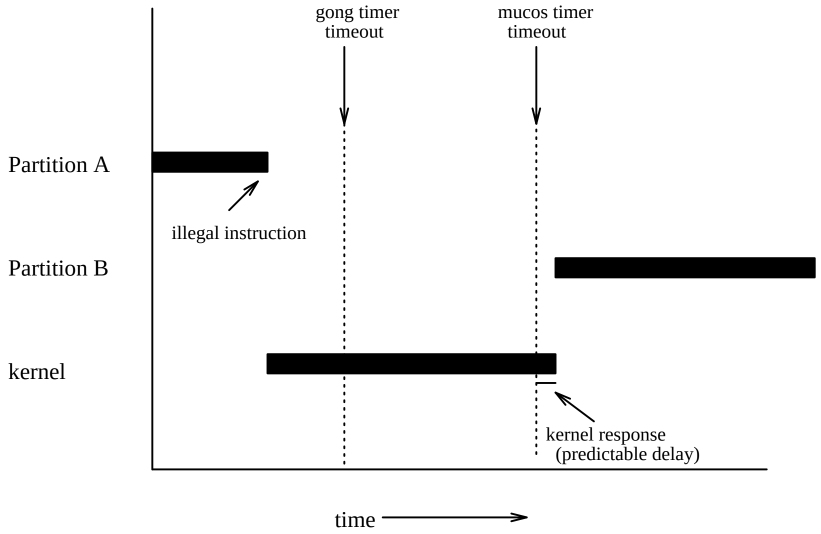

Fig. 7. Partition switch after illegal instruction

5 Schultz Invariant Performance

Figure 8 presents our PVS formalization of invariant performance for Schultz. Given two good initial Schultz states that have equal length schedules with a valid schedule element i that has identical base/limit values and processor memory that is identical for that region, then running the system will yield a state with identical memory in that region starting in either state.

```
schultzip: LEMMA
  FORALL (sys1,sys2: system_state) (i: validpart) (fin: nat):
  LET m1= memory(conns(sys1)), m2= memory(conns(sys2)) IN
    initial_schultz(sys1) AND initial_schultz(sys2) AND
    length(m1) = length(m2) AND i < length(m1) AND
    psdequal(i,m1,m2) AND psdvalid(i,m1) AND psdvalid(i,m2) AND
    memoryrange(m1,base(i,m1),limit(i,m1)) =
      memoryrange(m2,base(i,m1),limit(i,m1))
    IMPLIES
      memoryrange(memory(conns(protected_system(sys1,0,fin))),
                  base(i,m1),limit(i,m1)) =
        memoryrange(memory(conns(protected_system(sys2,0,fin))),
                    base(i,m1),limit(i,m1))
```

Fig. 8. Schultz invariant performance in PVS

The conjecture has four universally-quantified variables: `sys1` and `sys2`, initial system states, `i`, a schedule element, and `fin`, an ending time. The hypotheses of the conjecture restrict the applicability of the conclusion to "reasonable" initial states and schedule elements — the initial states have the kernel loaded, the schedule element must have identical corresponding memories in the initial states, the schedule element must actually be in the schedule, etc. The ending time at which the memories must correspond has no restriction — the conjecture must hold at all times during execution of the system.

We are currently proving this conjecture using the PVS theorem prover. There are two main lemmas needed to prove the invariant performance conjecture. The "reset" lemma guarantees that resetting the AAMP-FV with Schultz leads to a reasonable state. The "maintenance" lemma guarantees that the code in Figure 5 maintains various invariants about the system that can be used to guarantee invariant performance. We briefly sketch the proof of the maintenance lemma.

The maintenance lemma is similar to the schultzip lemma in Figure 8, except rather than assume that the two Schultz states satisfy "initial Schultz" conditions they will be assumed to have "reasonable timer values" and be "safely executing the current user partition". Reasonable timer values means that the mucos timer is greater than the gong timer value plus the maximum user instruction time plus the maximum handler execution time (excepting the write of the mucos register). Safely executing the current user partition means that the PMU is initialized with the current base/limit pair, the processor is in user mode, and the executive PSD is initialized correctly.

The proof of the maintenance lemma is by induction on the time required to complete one user partition and partition swap. The base case is when the `fin` value occurs before the completion of the partition swap to the next partition, which we prove by showing that the only way to modify partition i's memory is if partition i is the current partition, in which case the memory is changed identically by the two executions. The inductive case is when the `fin` value occurs after the switch to the next partition. We show that the execution of a partition and the partition swap code maintains each of the hypotheses of this conjecture, which together with an inductive hypothesis suffices to prove the result.

Figure 6 shows a typical partition swap, where the gong timer generates an NMI which (after whatever latency is associated with finishing the current user instruction) transitions the machine into executive mode. After setting up for the next partition the kernel then waits for the mucos timer timeout before starting the next partition. A second possibility exists for the timing of partition swapping, however: a partition can transition back to the kernel as the result of the execution of an illegal instruction, as suggested by Figure 7.

6 Summary

We are currently proving Schultz invariant performance using the PVS theorem prover. We will extend this approach to more complex partitioning systems, in particular those that allow inter-partition communication. We expect our approach to extend to this kind of system by treating the communication channel reads among the inputs to a partition.

Statements of formal correctness that do not use an abstract execution model are rare. One example is the self-consistency checking work, wherein for example the operation of a processor's pipeline is specified using the same pipeline with "NOP"s inserted into the instruction stream [9]. Another is the symbolic simulation work whose objectives include regression testing of an evolving design by comparing symbolic execution of generations of designs [8].

The KIT correctness theorem, and all theorems that use an abstract model to specify system behavior, are not especially useful for justifying the integration of embedded applications. We believe that, from the standpoint of dependable embedded application integration, invariant performance is the right property to guarantee for partitioning systems.

References

1. William Bevier. *A Verified Operating System Kernel.* PhD thesis, University of Texas at Austin, September 1987. Also available as ftp://ftp.cs.utexas.edu/pub/boyer/diss/bevier.ps.Z.
2. William R. Bevier. KIT: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1368–81, November 1989.
3. William R. Bevier, Warren A. Hunt Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.
4. William R. Bevier and William D. Young. Machine checked proofs of the design of a fault-tolerant circuit. *Formal Aspects of Computing*, 4:755–775, 1992.
5. Robert S. Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.
6. Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design – FMCAD*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
7. Colin Fidge, Peter Kearney, and Mark Utting. Formal specification and interactive proof of a simple real-time scheduler. Technical Report 94-11, Software Verification Research Centre, The University of Queensland, April 1994.
8. David A. Greve. Symbolic simulation of the JEM1 microprocessor. In *Formal Methods in Computer-Aided Design – FMCAD*, Lecture Notes in Computer Science. Springer-Verlag, 1998.
9. Robert B. Jones, Carl-Johan H. Seger, and David L. Dill. Self-consistency checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design – FMCAD*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
10. Patrick Lincoln and John Rushby. The formal verification of an algorithm for interactive consistency under a hybrid fault model. In Costas Courcoubetis, editor, *Computer-Aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
11. Steven P. Miller, David A. Greve, Matthew M. Wilding, and Mandayam Srivas. Formal verification of the AAMP-FV microcode. Technical report, Rockwell Collins, Inc., Cedar Rapids, IA, 1996.

12. Steven P. Miller and Mandayam Srivas. Formal verification of the AAMP5 microprocessor: A case study in the industrial use of formal methods. In *WIFT'95: Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, 1995. IEEE Computer Society.
13. J Strother Moore. *Piton – A Mechanically Verified Assembly-Level Language*. Kluwer Academic Publishers, 1996.
14. S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language (Beta Release)*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.
15. David M. Russinoff. A mechanically checked proof of IEEE compliance of the AMD K5 floating-point square root microcode. Available as www.onr.com/user/russ/david/fsqrt.html, August 1996.
16. David M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division, and square root algorithms of the AMD-K7 processor. Available at www.onr.com/user/russ/david/, January 28 1998.
17. Matthew Wilding. A mechanically verified application for a mechanically verified environment. In Costas Courcoubetis, editor, *Computer-Aided Verification – CAV '93*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
18. Matthew Wilding. *Machine-Checked Real-Time System Verification*. PhD thesis, University of Texas at Austin, May 1996. Also available as ftp://ftp.cs.utexas.edu/pub/boyer/diss/wilding.ps.Z.
19. Matthew M. Wilding. Robust computer system proofs in PVS. In C. Michael Holloway and Kelly J. Hayhurst, editors, *LFM97: Fourth NASA Langley Formal Methods Workshop*. NASA Conference Publication no. 3356, 1997. (http://atb-www.larc.nasa.gov/Lfm97/).

Acknowledgments: This work was supported at Rockwell Collins Inc. by the Defense Advanced Research Projects Agency, Arpa order D855. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Rockwell Collins, the Defense Advanced Research Projects Agency, or the US government.